|  |  |
| --- | --- |
| **CSCI 2100E Spring 2020** | Student Name: \_Tam Yi Ki\_ Student Id: \_1155126127\_\_ |

# Lab Exercise #8

|  |
| --- |
| This is the **last set** of normal lab exercises. Thank you very much for putting effort in this course for these weeks! In future weeks exercises would not be in coding format.  Please fill in the lab sheet and submit the completed Word doc file to blackboard.  Places you need to fill in or work on are marked in red. |

*The lab is due in three weeks after the lab date.*

## Problem 1 [Binary Expression Tree]

Below we have a very simple implementation of a binary expression tree. The In-order traversal is already done for you. Please fill in the proper code so that **pre-order traversal** and **post-order traversal** will work as well? Please consult the lecture notes on Binary Trees if you forgot about different traversals - the answers are (almost) written there.

If your code is right the output will be:

|  |
| --- |
| In-order: 3\*2\*4-1  Pre-order: \*3-\*241  Post-order: 324\*1-\* |

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  typedef struct node {  char key;  struct node \*left;  struct node \*right;  } Node;  Node\* createTree() {  // create an example tree at root  Node\* root = (Node\*) malloc(sizeof(Node));  root->key = '\*';    Node\* node1 = (Node\*) malloc(sizeof(Node)); node1->key = '3';  node1->left = node1->right = NULL;  root->left = node1;  Node\* node2 = (Node\*) malloc(sizeof(Node)); node2->key = '-';  node2->left = node2->right = NULL;  root->right = node2;    Node\* node3 = (Node\*) malloc(sizeof(Node)); node3->key = '\*';  node3->left = node3->right = NULL;  node2->left = node3;  Node\* node4 = (Node\*) malloc(sizeof(Node)); node4->key = '1';  node4->left = node4->right = NULL;  node2->right = node4;    Node\* node5 = (Node\*) malloc(sizeof(Node)); node5->key = '2';  node5->left = node5->right = NULL;  node3->left = node5;  Node\* node6 = (Node\*) malloc(sizeof(Node)); node6->key = '4';  node6->left = node6->right = NULL;  node3->right = node6;    return root;  }  void InOrderTraversal(Node\* node) {  if (node == NULL) return;  InOrderTraversal(node->left);  printf("%c",node->key);  InOrderTraversal(node->right);  }  void PreOrderTraversal(Node\* node) {  // TODO: implement  if (node == NULL) return;  printf("%c", node->key);  PreOrderTraversal(node->left);  PreOrderTraversal(node->right);  }  void PostOrderTraversal(Node\* node) {  // TODO: implement  if (node == NULL) return;  PostOrderTraversal(node->left);  PostOrderTraversal(node->right);  printf("%c", node->key);  }  int main()  {  Node\* root = NULL;    root = createTree();  printf("In-order: ");  InOrderTraversal(root);  printf("\nPre-order: ");  PreOrderTraversal(root);  printf("\nPost-order: ");  PostOrderTraversal(root);  printf("\n");    return 0;  } |

Using any tools, draw the binary tree in graphical form in the space below (i.e. use circles and arrows as in lecture notes)

|  |
| --- |
|  |

# Problem 2

The in-order traversal, as we have mentioned in the lecture, is misleading as it cannot be evaluated correctly without parenthesis. Let's add parenthesis! Update the InOrderTraversal function so that parenthesis will be added. For instance, the output for in-order traversal should now be:

|  |
| --- |
| In-order: (3\*((2\*4)-1)) |

Please paste your updated InOrderTraversal function here:

|  |
| --- |
| void InOrderTraversal(Node\* node) {    if (node == NULL) return;  if(node->left){  printf("(");  InOrderTraversal(node->left);  }  printf("%c",node->key);  if(node->right){  InOrderTraversal(node->right);  printf(")");  }    } |

# Problem 3

The pre-order traversal, as we mentioned in lecture, works very similar to how C/mathematical functions syntax works. So in the example in Problem 1,

|  |
| --- |
| In-order: 3\*2\*4-1  Pre-order: \*3-\*241  Post-order: 324\*1-\* |

The pre-order traversal can be written in this functional form:

|  |
| --- |
| Pre-order: mul(3,sub(mul(2,4),1)) |

Here mul is a function for multiplication, sub for subtraction. We can also have add for addition and div for division. All of them, obviously, two parameters.

Update the PreOrderTraversal so that it will give the functional form above. Please paste your updated PreOrderTraversal function here:

|  |
| --- |
| void PreOrderTraversal(Node\* node) {  // TODO: implement  if (node == NULL) return;  if(node->key == '\*')  printf("mul(");  else if(node->key == '-')  printf("sub(");  else  printf("%c", node->key);    PreOrderTraversal(node->left);  if(node->right){  printf(",");  PreOrderTraversal(node->right);  printf(")");  }  } |

# Problem 4 [Using Array to Implement Binary Tree]

In our lecture on Heap we mentioned that we can use an array to store a tree. How ridiculous! Let's investigate the **space complexity** of using an array to store a general binary tree, step by step.

First, what is the minimum and maximum height of a binary tree of **N** nodes?

|  |
| --- |
| Min = log(2,N)  Max = N - 1 |

If we use an array to store a binary tree, it is natural to see that we will need to consider the minimum/maximum height of the binary tree.So to allocate an array to store a binary tree of **N** nodes, what is the array size required if you consider the a) minimum height and the b) maximum height of the tree?

(To give you a fun fact, the sum of 1 + 2 + 4 + … 2n is actually 2n+1-1)

|  |
| --- |
| (a) 2N+1-1  (b) N(N-1)/2 |

From your result above, when would it be reasonable to store a binary tree in an array?

|  |
| --- |
|  |

# Problem 5 [Binary Search Tree]

We have also discussed binary search tree (BST) in the lecture. Here we see a BST example. The search function is provided as reference. Please **complete the insert function** (it's quite easy - it is very similar to the search function!). If done correctly the output of the program should become:

|  |
| --- |
| Original Tree: 023469  9 found!  4 found!  1 not found!  12 not found!  Tree after inserting 1,8: 012346789  Tree after deleting 11,8,3: 012346789  Tree after deleting 6: 012346789 |

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  typedef struct node {  int key;  struct node \*left;  struct node \*right;  } Node;  Node\* createTree() {  // create an example tree at root  Node\* root = (Node\*) malloc(sizeof(Node));  root->key = 6;    Node\* node1 = (Node\*) malloc(sizeof(Node)); node1->key = 2;  node1->left = node1->right = NULL;  root->left = node1;  Node\* node2 = (Node\*) malloc(sizeof(Node)); node2->key = 9;  root->right = node2;  node2->left = node2->right = NULL;  Node\* node3 = (Node\*) malloc(sizeof(Node)); node3->key = 0;  node3->left = node3->right = NULL;  node1->left = node3;  Node\* node4 = (Node\*) malloc(sizeof(Node)); node4->key = 3;  node4->left = node4->right = NULL;  node1->right = node4;    Node\* node6 = (Node\*) malloc(sizeof(Node)); node6->key = 4;  node6->left = node6->right = NULL;  node4->right = node6;    return root;  }  void InOrderTraversal(Node\* node) {  if (node == NULL) return;  InOrderTraversal(node->left);  printf("%d",node->key);  InOrderTraversal(node->right);  }  Node\* search(Node\* node, int key) {  // search the tree/subtree headed by "node" for "key"  if (node == NULL) return NULL;  if (node->key == key) return node;  else if (key < node->key) return search(node->left, key);  else if (key > node->key) return search(node->right, key);  }  void insert(Node\* node, int key) {  // insert the "key" into the right position  // inside the tree/subtree with "node" as root  // TODO: implement  if (node == NULL) {  return;  } else if (node->key > key) {  if (!(node->left)) {  Node\* insertNode = (Node\*) malloc(sizeof(Node));  insertNode->key = key;  node->left = insertNode;  } else {  insert(node->left, key);  }  } else if (node->key < key) {  if (!(node->right)) {  Node\* insertNode = (Node\*) malloc(sizeof(Node));  insertNode->key = key;  node->right = insertNode;  } else {  insert(node->right,key);  }  }  }  Node\* delete(Node\* root, int key) {  Node\* temp = (Node\*)malloc(sizeof(Node));  if (root == NULL)  return root;  if (root->key > key)  root->left = delete(root->left, key);  else if (root->key < key)  root->right = delete(root->right, key);  else{  if (root->left == NULL){  struct node \*temp = root->right;  free(root);  return temp;  }else if (root->right == NULL){  struct node \*temp = root->left;  free(root);  return temp;  }  root->key = temp->key;  root->right = delete(root->right, temp->key);  }  return root;  }  int main()  {  Node\* root = NULL;    root = createTree();  printf("Original Tree: ");  InOrderTraversal(root);  printf("\n");    // testing search  int queries[] = {9,4,1,12};  int i;  for (i=0;i<4;i++) {  Node\* result = NULL;  result = search(root,queries[i]);  if (result != NULL) printf("%d found!\n", result->key);  else printf("%d not found!\n", queries[i]);  }    // testing insert  insert(root,1);  insert(root,8);  insert(root,7);  printf("Tree after inserting 1,8,7: ");  InOrderTraversal(root);  printf("\n");    // testing delete  root = delete(root,11);  root = delete(root,7);  root = delete(root,3);  printf("Tree after deleting 11,7,3: ");  InOrderTraversal(root);  printf("\n");  root = delete(root,6);  printf("Tree after deleting 6: ");  InOrderTraversal(root);  printf("\n");  return 0;  } |

# Problem 6 (Hard, Optional)

Please complete the **delete** function now. There are four cases:

1. If the node is NULL, nothing needs to be done
2. If the node has no children, simply remove the link from parent and deallocate it
3. If the node has one child, replace oneself with the child
4. If the node has two child, please **replace with the min from the right subtree**

Please **consult the lecture notes** for an example. If done correctly the output of the program should become:

|  |
| --- |
| Original Tree: 023469  9 found!  4 found!  1 not found!  12 not found!  Tree after inserting 1,8,7: 012346789  Tree after deleting 11,7,3: 0124689  Tree after deleting 6: 012489 |

Paste your delete function below.

|  |
| --- |
|  |

|  |
| --- |
| Please finish other Lab Exercises if you haven't already! |